An Approach to Translate XSLT into XQuery

Albin Laga, Praveen Madiraju and Darrel A. Mazzari

Department of Mathematics, Statistics, and Computer Science
Marquette University
P.O. Box 1881, Milwaukee, WI 53201

Contact Author: Praveen Madiraju

Email: praveen@mscs.mu.edu

Phone : 414-288-6340

ABSTRACT

XML (eXtensible Markup Language) transformations and queries are crucial operations for interpreting XML databases. XSLT (eXtensible Stylesheet Language Transformations) is a prominent XML technology for these operations, but XQuery (XML Query Language) can query a broad spectrum of XML information sources, including both databases and documents. In this paper, we present our approach to translate XSLT to XQuery. We illustrate our approach as a set of rules or templates which translates an XSLT document into XQuery. We also discuss grouping operations for both XSLT and XQuery. Finally, we present the performance of XSLT versus equivalently transformed XQuery documents for varying sizes of documents and for different rules.

Keywords

XSLT, XQuery, Xpath, XML Databases

1. INTRODUCTION

XML is becoming as a standard for the exchange of many data sources. The reason for the use of XML databases for database integration is, principally, together with W3C recommended query languages, such as XPath [3] and XQuery [4], and other techniques like XSLT [9] and SOAP (Simple Object Access Protocol), that XML helps to overcome most technical heterogeneities of diverse databases. In addition, XML databases can be transmitted and stored as text files with schema information transported using XML schemas or Document Type Definition files, unlike relational databases, though there may be speed degradation with very large text based databases. In order to extract information from XML databases, one solution would be to extend a knowledge modeling tool by implementing a set of new classes or functions in a language such as Java, though this can be a rather difficult and time consuming task. Another solution would be to use XSLT or XQuery for transforming XML documents. An advantage of this approach is that, even though an XSLT is written independently of any programming language, it can be *executed* by a program written in most modern programming languages [7].

XSLT and XQuery were developed in close collaboration, and therefore there is a high degree of functionality similarity of the two languages. They share many common concepts, such as the underlying data model. They

both include the whole of XPath as a sublanguage, which supports a number of data types and a respectable function library [5, 8].

The processing engine in XSLT automatically goes through the document tree and applies templates as it encounters nodes. XQuery, on the other hand, requires the programmer to direct the process. XSLT, in one sense, is like a Report Generator in which there exists an implicit processing cycle, and the programmer just sets up the actions that are desired to occur when certain conditions are met. XSLT is loosely typed as it is flexible in handling conversions between nodes, strings and numbers, for the most part, transparently. XQuery is a typed language which uses the types defined by XML Schema, and will complain when datatypes don't match with each other. XQuery implements the FLWR (For, Let, Where, & Result) expression. XQuery creates a list of binding tuples with the FOR clause which are taken from an ordered forest. Subsequent statements are executed once, including any additional binding tuples producing a cross product of these bindings. The construction of optimal bushy trees for computing the cross product of a set of relations is NP-hard. One of the problems the query optimizer has to deal with is the ordering of joins.

XSLT generates a current node list, with its "for-each" instruction, which is taken from a node set (XPath) in document order. In XQuery, like in a language such as "C", the programmer is responsible for directing algorithms. XSLT uses the syntax of true XML, while XQuery is only XML-like. A main difference between the two languages at the syntactic level is the way modularity is handled. XSLT has an include/import mechanism whose detailed semantics discourage independent compilation of modules. Imported modules do not even need to be a valid standalone style sheets, e.g., they can contain references to global variables declared by their caller. The XSLT approach, with its use of import precedence to select the templates, and functions that are invoked at run-time, provides a mechanism for polymorphism (i.e., customization of styles heets) that is missing entirely from XQuery [11]. By contrast, the import module feature in XQuery is designed to allow library modules to be compiled independently and linked at run-time. In Figure 1, we present a functional comparison between XSLT and XQuery. XSLT 2.0 [9] and XQuery 1.0 [4] target different user communities; the former is for transformation of one XML document into another, while the later targets querying of XML documents.

	XSLT		XQuery	
	Advantages	Disadvantages	Advantages	Disadvantages
Processing	Auto engine through document tree	Limited control	Programmer Directed	More effort, more possible errors
Data typing	Loosely typed	Parsing difficulties	Typed, more types possible	More effort, more possible errors
Recursion	Inherently recursive; less code	Programmer Intensive	Can be incorporated	Verbose code

Figure 1. A functional comparison of XSLT and XQuery

As it stands now, XQuery 1.0 recently became an official W3C recommendation and XSLT 2.0 is an official recommendation of W3C. We need to be able to translate XSLT 2.0 (at least the ones which query) to XQueries. Hence, we need a tool to translate XSLT to XQuery. Also, it is predicted that most of the database systems will support XQuery, but may not implement XSLT. Furthermore, such a translation is needed for those XML database developers, who may understand XQuery but not XSLT. In our previous work [10], we proposed architecture and rules to translate an XSLT into XQuery. In this paper, we extend the work by formalizing the approach with an algorithm, and carrying out performance evaluation of our system.

The rest of the paper is organized as follows: In Section 2, we present our system architecture. We illustrate our approach for translating XSLT to XQuery with examples in Section 3. We then discuss grouping of data in both XSLT and XQuery in detail in Section 4. In Section 5, we present, *XSLT2XQ*, an algorithm for translating XSLT into XQuery using concepts discussed in previous sections. We also discuss our experiment results. Finally, in Section 6 we describe related work and present our conclusions.

2. SYSTEM ARCHITECTURE

The system architecture is shown in Figure 2. The XSLT to XQuery Translator takes as input XSLT 2.0 [9] and outputs an XQuery 1.0 [4]. As both XSLT 2.0 and XQuery 1.0 have XPath 2.0 as their subset, wherever equivalent methods exist in both of them, the application will utilize XPath 2.0 for transformations directly from XSLT 2.0 to XQuery 1.0. In the case where an equivalent method does not exist, functions that mimic the XSLT

2.0 methods will be generated for XQuery 1.0. The translator also uses the template rules as described in Section 3.1 to produce an XQuery document.

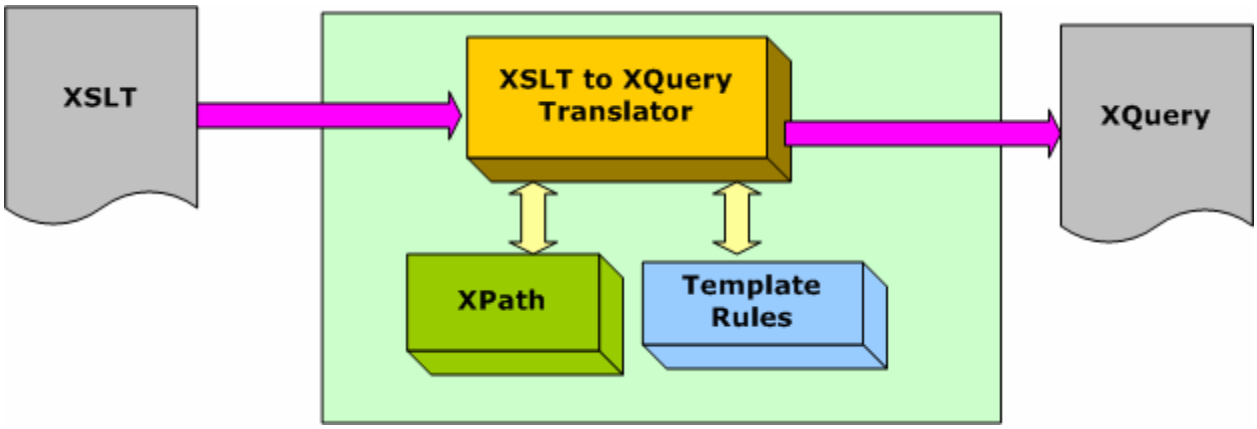

Figure 2. System architecture

XPath 2.0 is an expression language that allows processing of values conforming to the data model defined in the XQuery/XPath Data Model (XDM). The data model provides the XML documents in the form of a tree representation, and also as atomic values such as integers, strings, booleans, and sequences that may contain both references to nodes in an XML document or atomic values. The result of an XPath expression may be a selection of nodes from the input documents, or an atomic value, or, more generally, any sequence allowed by the data model.XPath navigates through the hierarchical structure of an XML document and hence the name [3].

3. OUR APPROACH

Throughout the paper, we illustrate our approach of translation using a sample XML document shown in Figure 3.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Edited with XML Spy v4.2 -->
<catalog>
  <cd>
      <title>Empire Burlesque</title>
      <artist>Bob Dylan</artist>
      <country>USA</country>
       <company>Columbia</company>
       <price>10.90</price>
       <year>1985</year>
  </cd>
  <cd>
       <title>Hide your heart</title>
```

```
        <artist>Bonnie Tyler</artist>
        <country>UK</country>
        <company>CBS Records</company>
        <price>9.90</price>
        <year>1988</year>
  </cd>
</catalog>
```

Figure 3. A fragment of cdcatalog.xml; a full version of the document is available from [15]

XPath is a rich language with many built-in functions included in it. XSLT and XQuery are closely related because the underlying path language for both XSLT and XQuery is XPath. Hence, the translation is much easier as they share the same built-in functions [5].

3.1 Translation Rules

XSLT relies on a highly declarative template-based approach, which gives ability to easily extend existing programs or merge programs together. On the other hand, XQuery is based on a purely functional approach, which gives more direct control to the user. We will show the translations of each XSLT construct into equivalent XQuery via the examples on the cdcatalog.xml (refer to Figure 3). We have tested for the correctness of all our XSLTs and XQueries using Saxon 8.7 [8].

Rule 1: XQuery and XSLT support declaration of parameters and variables. They can be declared and invoked in both languages. However, XQuery variables are more restricted because they have strict type.

XSLT	XQuery
`<xsl:param name="cd">Cd collection</xsl:param>` `<xsl:variable name="cds">My Cd collection</xsl:variable>`	`declare variable $c as xs:string := "Cd collection";`
Invoked by:	*calling a variable:*
`<xsl:value-of select="$cd"/>` `<xsl:value-of select="$cds"/>`	`Return <h2>{$c}</h2>`

Rule 2: The XSLT function **`<xsl:value-of>`** can be translated into a XQuery expression **`for $x in …(collection of data[1]).`** Since "**`for`**" clause returns a collection, we narrow the retrieved values to the very first one by explicitly asking for the first value of that collection **`[1].`**

XSLT	XQuery
`<xsl:value-of select="catalog/cd/title" />`	`for $x in …(collection of data[1])`

Invoked by:	*usage:*
`<xsl:value-of select="catalog/cd/title" />`	`for $x in doc("cdcatalog.xml")/catalog/cd[1]`

Rule 3: The XSLT function **`<xsl:for-each>`** can be translated into XQuery expression **`for $x in`**… as shown below:

XSLT	XQuery

`<xsl:for-each select="node">`
`<!–function Body -->`
`</xsl:for-each>`
	`For $x in …(collection of data)`

usage:	*usage:*

`<xsl:for-each select="catalog/cd">`
`<tr>`
`<td><xsl:value-of select="title"/></td>`
`<td><xsl:value-of select="artist"/></td>`
`</tr>`
`</xsl:for-each>`
	`for $x in doc("cdcatalog.xml")/catalog/cd`

Rule 4: Filtration is very straight forward in both languages and can be specified in the square brackets in an XPath expression. Alternatively, in XQuery we can simply use a "where" clause.

XSLT	XQuery

`XPath/expression/[where condition]/node`	`XPath/expression/[where condition]/node`
	`Or`
	`where variable = "something"`

usage:	*usage:*

`<xsl:for-each select="catalog/cd[artist='Bonnie Tyler']">`
`<tr>`
`<td><xsl:value-of select="title"/></td>`
`<td><xsl:value-of select="artist"/></td>`
`</tr>`
`</xsl:for-each>`
	`for $x in doc("cdcatalog.xml")/catalog/cd[artist= "Bonnie Tyler"]`

Rule 5: Sorting can be implemented in XSLT by the using the function **`<xsl:sort/>`** and this must be placed as the very first one in the statement designated to retrieve a set of values. In XQuery we use "**`order by`**" clause.

XSLT	XQuery
`<xsl:sort select="element"/>`	`Order by variable/XPath/expression`
usage:	*usage:*

`<xsl:sort select="artist"/>`	`Order by $x/artist`

Rule 6: The XSLT function **`<xsl:if>`** can be translated into XQuery by **`if () then`** `<!–function Body -->` **`else()`** function while the **`else()`** statement must be included with empty parentheses.

XSLT	XQuery

`<xsl:if test="conditional expression">`
`  <!–function Body -->`
`</xsl:if>`
	`If( conditional expression ) then`
	`  <!–function Body -->`
	`else()`

Usage:	*usage:*

`<xsl:if test="price > 10">`
`  <tr>`
`  <td>`
`  <xsl:value-of select="title"/>`
`  </td>`
`  <td>`
`  <xsl:value-of select="artist"/>`
`  </td>`
`  </tr>`
`</xsl:if>`
	`return  if($x/price  > 10) then`
	`  <tr>`
	`  <td>{data($x/title)}</td>`
	`<td>{data($x/artist)}</td>`
	`  </tr>`
	`  else()`

Rule 7: The XSLT function **`<xsl:choose>`** can translated into XQuery by the following function **`if() then`** `<!–function Body -->` **`else() if() then`** `<!–function Body -->` **`else()`** `<!–function Body -->.` We can have many **`<xsl:when>`** elements but only one **`<xsl:otherwise>`** element in a **`<xsl:choose>`** function. Similary, in XQuery we can have many **`if() then`** `<!–function Body -->` elements but only one **`else()`** `<!–function Body -->` element in a **`if() then`** function.

XSLT	XQuery

`<xsl:choose>`
`  <xsl:when test=" conditional expression ">`
`  <!–function Body -->`
`  </xsl:when>`
`  <xsl:when test=" conditional expression ">`
`  <!–function Body -->`
`  </xsl:when>`
`  <xsl:otherwise>`
`  <!–function Body -->`
`  </xsl:otherwise>`
`</xsl:choose>`
	`if($x/price  > 10) then`
	`  <!–function Body -->`
	`if ($x/price  > 9) then`
	`  <!–function Body -->`
	`else()`
	`  <!–function Body -->`

Usage:	*usage:*
<xsl:choose> <xsl:when test="price > 10"> <td bgcolor="#ff00ff"> <xsl:value-of select="artist"/> </td> </xsl:when> <xsl:when test="price > 9"> <td bgcolor="#cccccc"> <xsl:value-of select="artist"/></td> </xsl:when> <xsl:otherwise> <td><xsl:value-of select="artist"/></td> </xsl:otherwise> </xsl:choose>	return if($x/price > 10) then <tr> <td>{data($x/title)}</td> <td bgcolor="#ff00ff">{data($x/artist)}</td> </tr> if ($x/price > 9) then <tr> <td>{data($x/title)}</td> <td bgcolor="#cccccc">{data($x/artist)}</td> </tr> else()

Rule 8: Templates in XSLT can be translated into user defined functions in XQuery. This topic is covered in depth in [5].

XSLT	XQuery
<xsl:template match="node"> <!—Template Body --> </xsl:template>	declare function local:cd-info($cd as element()?) as element()? { <!—Function Body --> }
Invoked by:	*calling a variable:*
<xsl:apply-templates/> <xsl:apply-templates select="XPath/to/node "/>	local:cd-info($x)

Grouping of data can be achieved in both XSLT and XQuery. We illustrate these operations in the next section. Thus the application presents a general method for translating the highly declarative rule-base approach of XSLT into the purely functional XQuery approach, leading the way to closer integration between the two languages.

4. GROUPING IN XSLT and XQuery

Grouping of data is a sophisticated operation. Here, we show grouping in XSLT and then an equivalent way of implementing it in XQuery

4.1 Grouping in XSLT 2.0

Here, we explain the process of grouping data from an XML document in detail using XSLT. There is a way to implement grouping in XSLT 1.0, however, it is a roundabout way of implementation of grouping [13].

Fortunately, starting from version 2.0 of XSLT, grouping has been simplified by introducing `xsl:for-each-group` element dedicated function for this purpose (refer to Figure 4).

```
<xsl:for-each-group
       select = "expression"
       group-by? = "expression"
       group-adjacent? = "expression"
       group-starting-with? = "pattern"
       group-ending-with? = "pattern"
       collation? = { uri }>
<!– Content: (xsl:sort*, sequence
constructor)   -->
</xsl:for-each-group>
```

Figure 4: Syntax of <xsl:for-each-group> element

The `xsl:for-each-group` creates a list (sequence) of items to be grouped, called *population*. The criteria for creating the *population* is provided by select attribute. Then, the list of distinct values of "*Grouping keys*" is created, and the criteria for creation of this list are provided by two attributes: group-by and group-adjacent.

The assignment of population items to groups depends on the group-by, group-adjacent, group-starting-with, and group-ending-with attributes based either on common value of a *grouping key*, or on a *pattern*.

The *population* can be grouped in four different ways depending on which of the four grouping attribute are provided: group-by, group-adjacent, group-starting-with, or group-ending-with. Only one of those attributes can be provided at a time, no more or less. In Figure 5, we show an example XSLT which selects artist and title information grouping by each country.

The return value of xsl:for-each-group is a *sequence of groups*. This feature makes this function very efficient since it returns a sequence of groups at one time. This is one of the main reasons why grouping in XSLT 2.0 works faster than grouping in XQuery 1.0 (see performance results in Section 5.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="catalog">
  <html>
   <body>
    <h2>My CD Collection</h2>
    <ul>
     <xsl:for-each-group select="cd" group-by="country">
      <b>
       <xsl:value-of select="country"/>
      </b>
      <xsl:for-each select="current-group()">
       <p>
```

```
        <xsl:apply-templates select="title"/>
        <xsl:apply-templates select="artist"/>
       </p>
      </xsl:for-each>
      <p>CD# = <xsl:value-of select="count(current-group()/title)"/>
      </p>
     </xsl:for-each-group>
    </ul>
   </body>
  </html>
 </xsl:template>
 <xsl:template match="cd">

 </xsl:template>
 <xsl:template match="title">
  Title: <span style="color:#ff0000">
   <xsl:value-of select="."/>
  </span>
  <br/>
 </xsl:template>
 <xsl:template match="artist">
  Artist: <span style="color:#00ff00">
   <xsl:value-of select="."/>
  </span>
  <br/>
 </xsl:template>
</xsl:stylesheet>
```

Figure 5: An example XSLT illustrating grouping operation

4.2 Grouping in XQuery 1.0

There is no dedicated function for grouping in XQuery 1.0. In order to group data in XQuery 1.0, we need to follow these two steps:

1) Create the list of distinct values (grouping keys) by which we want to group our data, and save the result in a sequence variable.

2) In the *return* statement, place the sequence variable from the earlier step and then, use the current value of this variable as an argument for the *where* clause of the next loop statement. This statement returns the group element that satisfies the condition of *where* clause.

If we want to use an aggregation function, XQuery has to construct the group again. The example in Figure 6 is an equivalent transformed XQuery 1.0 document for the XSLT shown in Figure 5.

```
(: declare functions :)
declare function local:cd-info($cd as element()? ) as element()?
{
  let $t := $cd/title
  let $a := $cd/artist
  return
    <p>Title:<span style="color:#ff0000">{data($t)}</span>
        <br/>
```

```
        Artist:<span style="color:#00ff00">{data($a)}</span>
    </p>
};
<html>
<body>
<h2>Cd Collection</h2>
{

for $c in distinct-values(doc("cdcatalog.xml")
/catalog/cd/country)
return
    <p>
      {$c}
      {
      for $x in doc("cdcatalog.xml")/catalog/cd
      where $x/country = $c
      return <li>{local:cd-info($x)}</li>
      }
      CD#= {count(doc("cdcatalog.xml")/catalog/cd
[country = $c])}
      <p/>
    </p>
}
</body>
</html>
```

Figure 6: An equivalent transformed XQuery for the XSLT in Figure 5

In Figure 6, the second loop of the *return* statement has to re evaluate as many times as the number of elements in variable $c. Every time the *return* statement loops, it creates a new group, and this process increases the execution time of XQuery and makes it less efficient than XSLT 2.0. We can say that XSLT 2.0 creates the **sequence of groups** only once, while XQuery creates the group as many times as the *return* statement loops. This is one of the main reasons why XQuery 1.0 is slower in comparison to XSLT 2.0.

5. ALGORITHM AND PERFORMANCE EVALUATION

Here, we give the algorithmic steps needed for translating an input XSLT document into an XQuery. The algorithm is given below.

```
Algorithm XSLT2XQ

1: Input – An XSLT document D_XSLT

2: Output – An XQuery document D_XQuery

3: let D_XQuery := null

4: for each template, T_XSLT in D_XSLT do

5:    for each rule i in {1 … 8} and grouping rule do
```

```
6:          if (T_XSLT matches rule(i)) then
7:                  D_XQuery = D_XQuery + (apply-rule(i) to T_XSLT )
```

Algorithm *XSLT2XQ* takes as input an XSLT document, D_{XSLT}, and generates an equivalent XQuery document, D_{XQuery}. For each template, T_{XSLT} in D_{XSLT}, we check if it matches with one of the eight template rules discussed in Section 3 or the grouping rule discussed in Section 4. If it matches, we apply the appropriate rule to the template T_{XSLT}, and this process repeats for all the templates in the input XSLT document. Consider, n as the total number of templates in D_{XSLT}. The first for loop in line 4 of the algorithm iterates over all the n number of templates, and the second for loop in line 5 of the algorithm iterates over all the rules, which is a constant. Hence, the overall time complexity is $O(n)$. In most cases, n is not a huge number, hence the overall time complexity can be considered as $O(1)$.

5.1 Performance Evaluation

We conducted experiments on the execution times for an input XSLT 2.0 versus an equivalently transformed XQuery 1.0. We run all experiments on the same 2.8 GHz Pentium 4 machine with 512 MB memory and one hard disk with 7200 rpm running Windows XP. We perform experiments on the cdcatalog.xml [15] dataset discussed before.

We generate synthetic XML data set using the sun multi-schema XML generator, which is part of sun's XML validator [14]. We use this tool to generate varying file sizes based on the cdcatalog XML schema. Interested readers can refer to the source code, data set and scripts for running the samples at [10]. We also note that (i) we have not yet fully conducted performance evaluations against all the benchmark XSLT test cases, and (ii) XQuery 1.0 candidate recommendation has been recently finalized by W3C, so better optimizations and new functions could be introduced in the future.

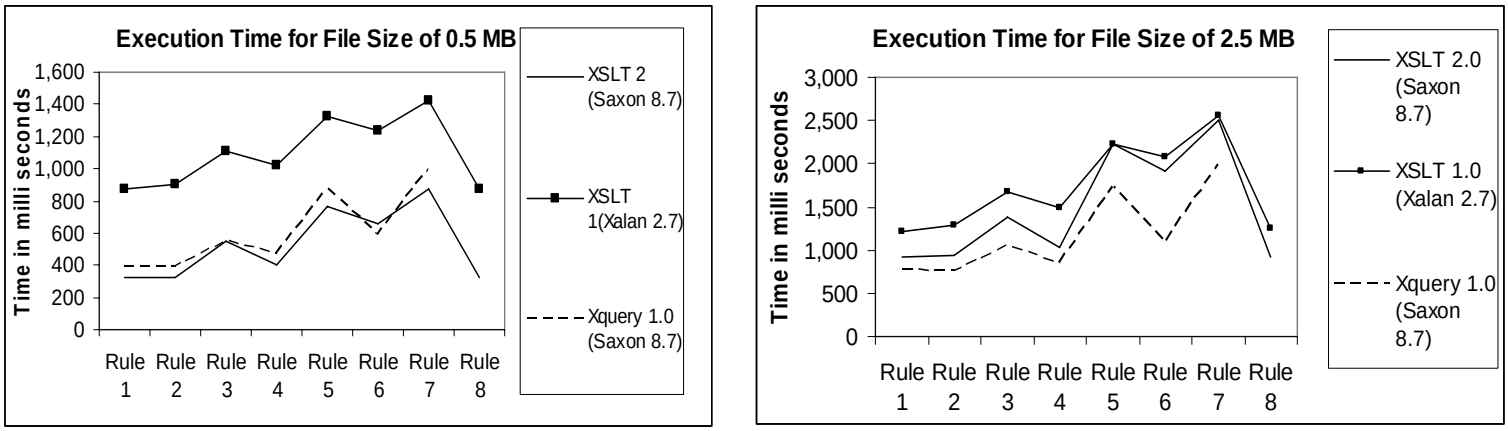

Figure 7 (a) and (b) : Execution time for file sizes 0.5MB and 2.5MB; note that XQuery 1.0 for Rule 8 in (a) is 380,000 milli sec, and the same in (b) takes 3,900,000 milli sec (not shown in the figure)

Figure 8: Execution time for grouping operation

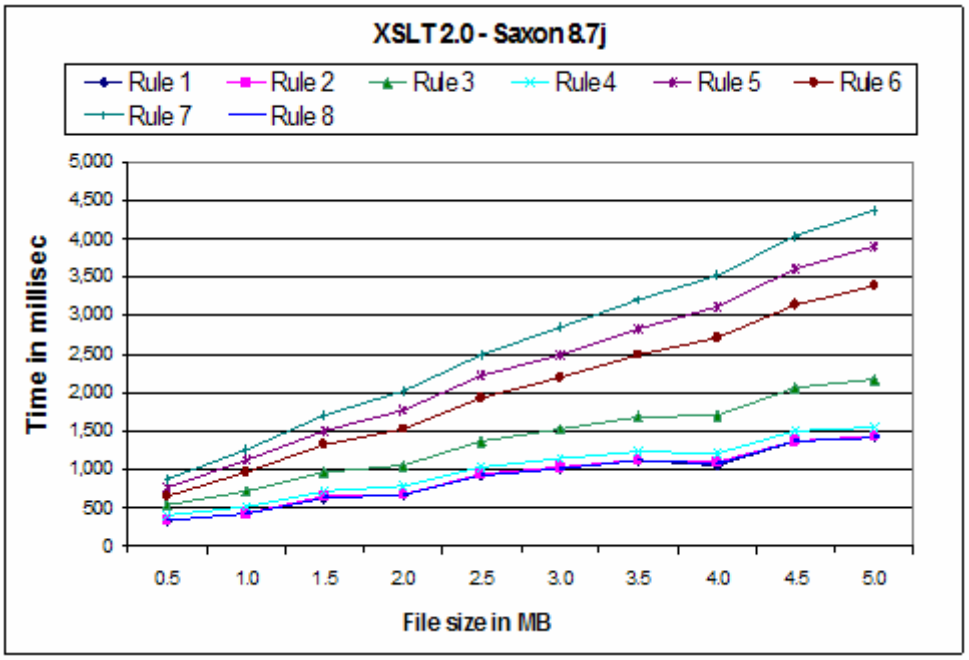

Figure 9: Execution time for XSLT 2.0

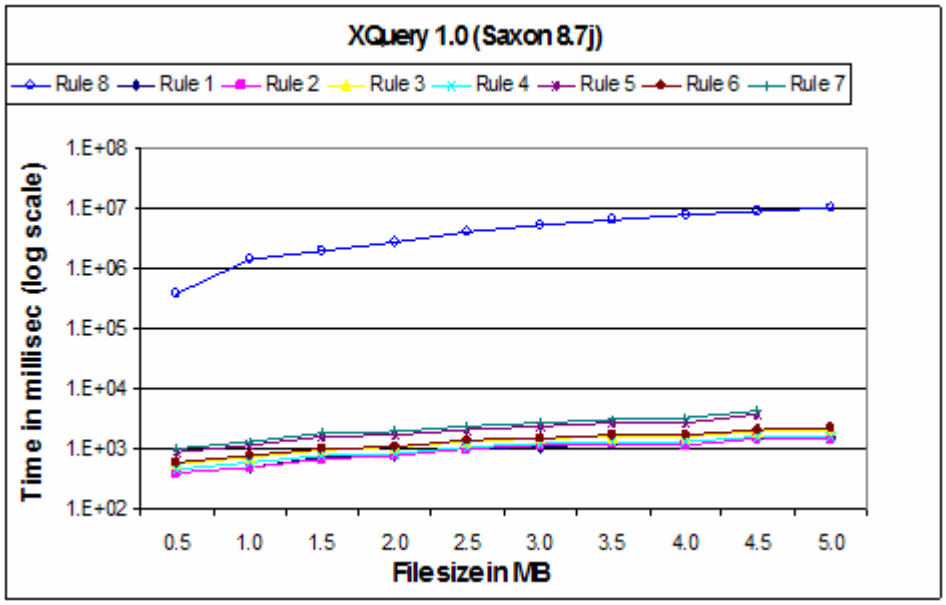

Figure 10: Execution time for XQuery 1.0

We evaluate the performance of both XSLT and XQuery documents generated as a result of applying (i) rules 1,2,…8 discussed in Section 3.1, and (ii) grouping operation discussed in Section 4.0 using three different softwares : Xalan 2.7 with XSLT 1.0, Saxon 6.5 with XSLT 1.0, Saxon 8.7 with XSLT 2.0, and Saxon 8.7 with XQuery 1.0. We do these evaluations for different files sizes ranging from 0.5 MB to 5.0 MB. It should be noted that performance of XSLT 1.0 with Saxon 6.5 and XSLT 2.0 with Saxon 8.7 were very identical. Hence, we skip them in the charts shown above. We present the execution times for file sizes 0.5 MB and 2.5 MB in Figures 7(a) and 7(b). As the file size increases, we note that XQuery execution time gets better. The only exception is the abnormal behavior of XQuery 1.0 for Rule 8. Saxon executes XQuery user defined functions very slowly; however, other XQuery interpreters show better performance. But, for a fair comparison, between XQuery and XSLT, we only show saxon's behavior, as the other software packages only supported either XQuery or XSLT with the exception of some commercial software packages. Also, as noted in Figure 8, XQuery performs poorly for grouping operation as the size of the file increases. This behavior is due to the reasons explained in Section 4.2 and additionally because of the bindings of the tuples generated from the FOR clause. In Figure 9, we show execution time of XSLT 2.0 using Saxon 8.7j for all the rules under varying loads of file sizes. Similarly, in Figure 10, we present execution time of XQuery 1.0 using Saxon 8.7j for all the rules under varying loads of file sizes.

6. RELATED WORK AND CONCLUSIONS

Related Work: There are many software packages available in the market such as [8, 16], which support both XSLT and XQuery executions. None of the commercial software packages support automatic translations. Fokoue et al. [5] present a detailed overview of compiling an XSLT 2.0 document into XQuery 1.0 document. The theoretical treatment especially is clear. However, the authors do not address grouping and sorting of XSLT's into XQueries.

Bezivin et al. [2] describe an approach to transform XSLT into XQuery using Atlas Transformation Language (ATL) within the Model Driven Architecture (MDA). They present more emphasis on the language details and do not go into details about the transformation approach with examples such as grouping and others. However, we

present our approach in a simple form with clear examples all throughout the paper. The authors in [11] present translation from XSLT to SQL queries. Moerkotte (2002) [12] describes an implementation to transform XSLT to database algebra.

Conclusions: We have presented a general framework and an approach to translate XSLT 2.0 to XQuery 1.0. We have illustrated our approach with many examples. As XQuery becomes more popular and attracts querying for XML documents, our tool will help XML database developers translate current XSLT documents to XQuery. We have also presented performance analysis of the different sets of XSLT documents versus XQuery documents on various software packages. The experiments show in some cases especially for implementing grouping operations, XQuery performs poorly, and should be taken into account.

This study gives some open problems such as XQuery optimizations, and improving grouping functions in XQuery. Efficient indexing, compression and storage techniques will also improve performance of XQuery.

7. REFERENCES

[1] Apache Xalan. http://xml.apache.org/xalan-j/, last accessed on March 28, 2006

[2] Bézivin, J., Dupé, G., Jouault, F., Pitette, G. and Rougui, J.E. “First experiments with the ATL model transformation language: Transforming XSLT into XQuery”, OOPSLA 2003 Workshop, Anaheim, California.

[3] Berglund A., Boag S, Chamberlin D, Fern´andez M. F., Kay M, Robie J, and Sim´eon J. (2004). “ XML path language (XPath) 2.0”. W3c working draft, World Wide Web Consortium, July 2004.

[4] Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., and Simeon, J. “XQuery 1.0: An XML query language “. W3c working draft, World Wide Web Consortium, July 2004. http://www.w3.org/TR/2004/WD-xquery-20040723

[5] Fokoue, A., Rose, K., Siméon, J., and Villard, L. “Compiling XSLT 2.0 into XQuery 1.0 “. In Proceedings of the Fourteenth International World Wide Web Conference, ACM Press, Chiba, Japan, May 2005, pp. 682-691.

[6] Jain, S., Mahajan, R., and Suciu, D. 2002. “Translating XSLT programs to Efficient SQL queries”. In Proceedings of the 11th international Conference on World Wide Web (Honolulu, Hawaii, USA, May 07 - 11, 2002). WWW '02. ACM Press, New York, NY, 616-626.

[7] Jovanovic, J., and Gasevic, D. “Achieving knowledge interoperability: An XML/XSLT approach”. Expert Systems with Applications, Volume 29, Issue 3, October 2005, Pages 535-553.

[8] Kay, M. (2005). “Comparing XSLT and XQuery”. Proceedings of the XTech 2005 Conference on XML, the Web and beyond ; software available from : http://saxon.sourceforge.net/, last accessed on March 29, 2006

[9] Kay, M. XSL transformations (XSLT) version 2.0. W3c working draft, World Wide Web Consortium, November 2003. http://www.w3.org/TR/2003/WD-xslt20-20031112.

[10] Laga, A., Madiraju, P., Mazzari, D.A.,and Dara, G. “Translating XSLT into XQuery”. Proceedings of 15th International Conference on Software Engineering and Data Engineering (SEDE-2006), Los Angeles, California, July 6-8, 2006. Scripts and code downloads available from : http://www.mscs.mu.edu/~praveen/Research/XSLT2XQ/

[11] Liu, J., and Vincent, M. “Query Translation from XSLT to SQL”. Seventh International Database Engineering and Applications Symposium (IDEAS),2003.

[12] Moerkotte, G. “Incorporating XSL processing into database engines”. In VLDB , September, 2002, Hong Kong, China, pp. 107–118.

[13] Tennison, J. “Grouping Using the Muenchian Method”. http://www.jenitennison.com/xslt/grouping/muenchian.xml, last accessed December 12, 2005

[14] The Sun Multi-Schema XML Validator(MSV). https://msv.dev.java.net/, last accessed on March 28, 2006

[15] W3 School, XML Application. http://www.w3schools.com/xml/cd_catalog.xml, last accessed on December 12, 2005.

[16] XML Spy. XML Spy 2005. http://www.altova.com/products_ide.html, last accessed on March 28, 2006